Implementing a Physician's Workstation using Client/Server Technology and the Distributed Computing Environment

Thuan Q. Pham, M.S.[1], Charles Y. Young, Ph.D.[1], Paul C. Tang, M.D.[2], Henri J. Suermondt, Ph.D.[1], Jurgen Annevelink, Ph.D.[1]

[1]Hewlett-Packard Laboratories, Palo Alto, CA
[2]Northwestern Memorial Hospital, Chicago, IL

Abstract

PWS is a physician's workstation research prototype developed to explore the use of information management tools by physicians in the context of patient care. The original prototype was implemented in a client/server architecture using a broadcast message server. As we expanded the scope of the prototyping activities, we identified the limitations of the broadcast message server in the areas of scalability, security, and interoperability. To address these issues, we reimplemented PWS using the Open Software Foundation's Distributed Computing Environment (DCE). We describe the rationale for using DCE, the migration process, and the benefits achieved. Future work and recommendations are discussed.

INTRODUCTION

The objective of the Physician's Workstation (PWS) project [1] is to investigate and develop a comprehensive and highly integrated set of information management tools for use by physicians in ambulatory care. We successfully developed and deployed an experimental prototype using an open systems architecture [2] based on HP's broadcast message server (BMS) [3]. Although the prototype works well in the context of a small work group, it has several limitations due to the characteristics of the BMS framework: scalability is limited due to the narrow bandwidth of the BMS; security is limited; and interoperability is limited to only a few hardware platforms. To overcome these limitations, we redesigned and reimplemented PWS's distributed computing framework using the Distributed Computing Environment (DCE) [4] from the Open Software Foundation (OSF).

This paper describes the design and implementation of the new PWS prototype. In particular, we illustrate our leveraging of DCE's technologies and services to make PWS a more scalable, secure, robust, and open distributed application.

In the following sections, we first describe the original architecture of PWS and discuss its limitations with respect to scalability, security, and interoperability. We then present the new client/server architecture and DCE-based implementation which overcome those limitations. Finally, we conclude with a vision of future work with regard to the emerging distributed computing technologies.

BACKGROUND: ORIGINAL PWS ARCHITECTURE

The original PWS architecture, illustrated in Figure 1, uses a message server as a backbone for both communication and data integration. The strength of this architecture is that, using the simple string-based message protocol, new applications could be developed independently and integrated easily into the PWS environment. As a result, the original prototype achieved the desirable openness and integration among application components.

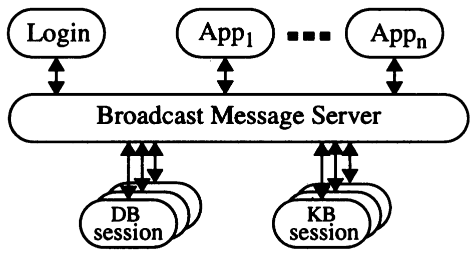

Figure 1: Original architecture of PWS that relies on BMS for communication and integration.

However, the architecture's total dependence on the broadcast message server has some undesirable consequences. Due to the limited capabilities of the BMS, the prototype lacked the scalability, security, and hardware interoperability needed for robust distributed applications. We address these issues in the following subsections.

Scalability

The scalability limitation of the original prototype is a result of the total reliance on the BMS as an all-purpose software bus for sending both control messages and large chunks of patient data among applications. All software components of PWS are dependent on one message server for all message distribution and data exchange. As more and more applications or sessions come on-line, the message server quickly becomes the bottleneck that degrades performance.

Security

The BMS environment does not provide any security services. Without a security service, an application cannot verify the identity or trust the intention of other applications. There is no provision in the BMS environment to prevent unauthorized users from receiving data that are placed on the bus, since anyone can subscribe to, receive, and monitor any and all bus traffic.

Furthermore, BMS messages are broadcast in plain text. Data encryption is needed, but it requires a security service component to authenticate and/or broker the encryption keys. Although it could be possible to implement a BMS-based security service, such effort would require a significant amount of work and would likely result in a non-standard implementation.

Interoperability

Currently, all PWS applications run on one hardware platform. To deploy the system on a much larger scale, PWS components will have to operate on a variety of hardware platforms, many of which may not support BMS.

NEW PWS ARCHITECTURE

Applications in the BMS environment can take on a client or server role in their behavior, but strictly speaking, these applications are peers that communicate by broadcasting on the BMS. From our experience in using PWS, we observe that most messages are one-to-one communications or data exchanges between two components. A more effective architecture would allow application components to communicate directly with each other, while using the message server to broadcast important events and shared information.

Thus, the fundamental goals of our new PWS architecture are to: use client/server technology to facilitate the direct communication between program components, use the event service mechanism for global control and event notification, and leverage DCE services to make PWS a robust, scalable, and secure distributed application.

What is DCE?

Developed by the Open Software Foundation (OSF), DCE is a framework and environment for building distributed applications in a heterogeneous computing environment. DCE provides both a development and a runtime environment for distributed applications. This includes the interface definition language (IDL) and compiler, remote procedure calls (RPC) [5,6], threads, directory and security services, distributed file system (DFS), distributed time service (DTS), and an application programming interface (API).

Why DCE?

The move to DCE enables us to leverage its distributed computing services and infrastructure to build robust distributed applications. For example, DCE's IDL allows us to define the interface between client and server applications; RPC makes a remote server's service appears like a local procedure call, greatly simplify networking; directory, timing, and security services are essential to any distributed system; and the emerging DCE standard and compliance ensure interoperability across many hardware platforms, ranging from workstations to personal computers.

PWS Client/Server Architecture

Figure 2 illustrates the new architecture of PWS, in which each application is either a client, a server, or both (since a server application can be a client of another). The interactions between programs are mostly in the form of a client invoking a service of a remote server using an RPC.

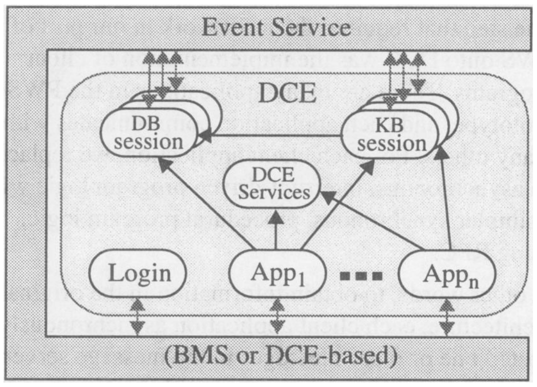

Figure 2: DCE-based client/server architecture of PWS

With RPC facilitating the direct data exchange between client and server applications, the event service is used effectively to handle control messages, such as broadcasting shared states and event notifications. For example, when a user logs in to PWS, the *Login* application creates a database session server, and broadcasts the server's full path name in the DCE environment to all interested application components. Using this name, applications can go to the DCE's cell directory service (CDS) to find and establish their own direct connections to the server. With a much lighter work load, the event service now has sufficient processing capacity to handle many more processes.

IMPLEMENTATION

Interface Definition

The first step in our implementation was to identify operations to be supported by server applications. From these, we defined the server interface using the IDL. The IDL compiler then generated the stub code for client and server modules. These stubs, when linked into the respective programs, enabled server and client applications to communicate with each other using the defined interface and RPC.

Server Implementation

To implement the server applications, we first implemented the server functionality as defined in the interface. Then, we wrote the server initialization and clean up routines so that the servers properly registered and unregistered themselves with DCE directory service. In addition, we set up an exception handling mechanism in each server so that it could remove itself from the DCE environment should an exception occur.

Client Implementation

The step that required the most work in our port of PWS onto DCE was the implementation of client programs. There are many applications in the PWS prototype, and each application communicates with many others. For each client application, we replaced an asynchronous, interrupt-driven program logic with a simpler synchronous, procedural program logic, using RPC.

In other words, to obtain information in the original architecture, each client application asynchronously posted one or more messages to the message server. When some other component provided an answer, the client application received the message from the message server, matched it with the originating request, and called some predefined functions to process the result. In the new architecture, most communications are point-to-point between the client and server components, in the form of (synchronous) remote procedure calls. With RPC, the results are available immediately when the call completes, removing the complexity of matching results with requests in the asynchronous mode of communication.

In addition, we enhanced our client applications by using multiple threads to implement parallel control. The use of threads in conjunction with RPC is significant because threads allow client applications to execute many RPC calls independently and concurrently. Hence, if an RPC call running in one client application thread is waiting for a server to produce the result, other threads in the client application can still proceed with their execution. Furthermore, by dispatching threads to carry out a user's requests, the main application thread is always under the user's control, thereby increasing the application's responsiveness.

For example, two of the most lengthy operations in PWS are the queries of a patient's detailed history of problems and medications, which involve the database server. When the provider first selects a patient to review, the display application must fetch patient information from the database while performing other tasks. To improve performance, we devised two job queue data structures, DxQueue and RxQueue, and assigned each queue to be processed by a thread. When the display application needs to fetch a patient's drug information or diagnosis, it simply places the query statements in the respective queue, and continues to handle more user requests. Meanwhile, for each query placed in the queue, the thread managing the job queue makes an RPC to get the results from the database.

Encapsulation of Non-DCE Components

In a new system, there may be a need to integrate legacy components. For this, DCE servers can be used to interface DCE components with non-DCE components. For example, the DCE/DB interface component in Figure 3 is actually a DCE server process that handles requests for data from a non-DCE database.

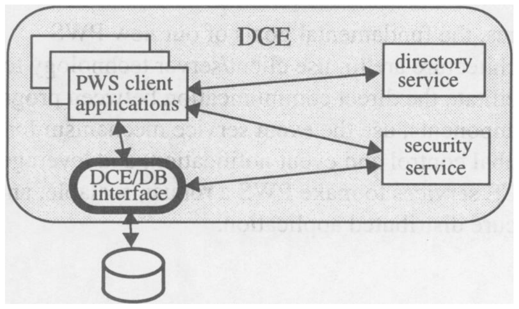

Figure 3: interfacing with non-DCE components

In our current implementation, we use a DCE server process as an interface to an object oriented database. When the DCE/DB interface server receives an RPC requesting information from a client, it queries the database. The query results are then returned to the calling process via an RPC return argument. Interface components such as this enables the integration and continued support of legacy systems.

DISCUSSION

The benefits of our migration to DCE are numerous. First of all, the new prototype is more scalable and robust. By allowing application programs to interact with one another directly, we removed the bottleneck from the broadcast message architecture. By using RPC to transmit data between client and servers, we achieved better efficiency and data privacy. By using the DCE directory service and RPC facility, we allowed client applications to interact with server applications without having to know their dynamic physical location in a distributed environment, and without having to handle the complex networking details. With DCE's RPC and services, accessing a server process running across the country was no more difficult than accessing one running on a local machine.

Furthermore, once ported, PWS applications immediately benefited from the security mechanism provided by DCE. These benefits include authentication, access control, and encryption.

The move to DCE also presented some challenges. First and foremost, the DCE environment requires system administration skills much like that of a Unix workstation (e.g., add new user account and security credential, setup user profile, etc.). The current lack of DCE system administration tools continues to make this task tedious and laborious.

In addition, DCE server processes are usually several megabytes in size and require much computing resources to run with adequate performance. In every DCE environment, there is an overhead of computing resources to run the essential DCE services such as security service, directory service, distributed time service, and various other essential DCE daemons.

DCE is a large and complex system. The DCE's architecture, facilities, capabilities, and application programming interface (which includes over several hundred functions) required some time to master.

To port an existing application to DCE, a change in the program's design is sometimes necessary. This is particularly true for applications that use asynchronous communication. Although RPC alleviates the problem of matching replies with requests, it is a synchronous communication protocol. Used serially, RPC takes away the program's ability to perform other work while the request is being processed by some server. This is unnecessarily restrictive and inefficient. Thus, to use RPC and still preserve the efficiency of asynchronous programs, we needed to redesign programs to use a combination of threads and RPC. In effect, we had to write multithreaded programs.

Lastly, although DCE is being accepted by more and more companies as the platform to implement open, distributed heterogeneous computing [7], its technology is still young. Early commercial implementations of DCE may still be a bit unstable and inefficient.

FUTURE EXTENSIONS

During the development of this PWS prototype, we identified a number of potential areas for further investigation.

Reduced Development Cost

Although building a robust DCE application is a complex job, the task is highly repetitive, and template driven. For example, a DCE server basically needs some initialization routines, clean-up routines, exception handling threads, and access control mechanism. After developing the first robust server, the task of implementing the basic structure of additional servers became routine.

OODCE [8], a C++ class library that encapsulates many DCE facilities, has been developed. This library offers much promise in reducing development cost and time. For example, an OODCE's server class object includes a full implementation of an exception handler, an access control list (ACL) manager, and some value-added facilities such as an object factory and activation mechanism. We would like to employ OODCE to help improve any future development.

Improved Fault-tolerance

As the system scales up to handle more and more processes, the reliance on a single message server creates a single point of failure. To address this, we can replace the message server with a hierarchy of DCE-based event services, which can be replicated to improve fault tolerance. Furthermore, such a DCE-based event service mechanism can interoperate with client programs from all DCE environments, making the system even more open.

Object Interoperability

The objects and distributed computing community is currently proposing a distributed object model that provides interoperability between different software platforms. For example, with such a model, an object in a Smalltalk application can interact with another object in a C++ application.

Specifically, the object model being adopted by the industry is the Common Object Request Broker Architecture (CORBA) [9,10] by the Object Management Group (OMG). To this extent, we ported a database interface component from DCE IDL to CORBAL IDL, and ported an application component, the drug formulation browser, of PWS to access the drug formulation server using the CORBA model. This work begins to establish a migration path from DCE to CORBA should we need to migrate in the future.

SUMMARY

We started this paper with a description of the original PWS architecture which relied on the asynchronous, broadcast message server as an integration framework for both interprocess communication and data exchange. In the discussion, we enumerated the architectural limitations that were learned through our use of PWS.

To overcome the limitations of the original architecture, we presented a new PWS design. The new architecture combines an efficient mix of synchronous, point-to-point client/server communication for data exchange, and asynchronous, broadcast messaging for event notification and operational synchronization. By using each communication model for its strength, the new PWS prototype is more efficient and scalable.

The scalability, security, and interoperability of a distributed application depend not only on the architecture, but also on the framework on which it is implemented. Using DCE, the new PWS prototype is able to leverage the distributed system technologies and services to achieve these properties.

ACKNOWLEDGMENTS

We would like to thank our colleagues Mike Higgins, Mark Gisi, Danielle Fafchamps, Phil Strong, Philippe De Smedt, and the reviewers for their insightful comments on previous drafts of this paper.

References

[1] P.C. Tang, J. Annevelink, D. Fafchamps, W.M. Stanton, and C.Y. Young. Physician's Workstations: Integrated Information Management for Clinicians. In: P. Clayton, ed., Proceedings of the Fifteenth Annual Symposium on Computer Applications in Medical Care. McGraw-Hill, New York, 1991, pp. 569-573.

[2] C.Y. Young, P.C. Tang, and J. Annevelink. An Open Systems Architecture for Development of a Physician's Workstation. In: P. Clayton, ed., Proceedings of the Fifteenth Annual Symposium on Computer Applications in Medical Care. McGraw-Hill, New York, 1991, pp. 491-495.

[3] M. Cagan. The HP Softbench Environment: an Architecture for a New Generation of Software Tools. Hewlett-Packard Journal, June 1990, pp. 36-47.

[4] OSF DCE Application Development Reference, Open Software Foundation, Revision 1.0, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[5] B.J. Nelson. Remote Procedure Call. Technical Report CSL-81-9, Xerox Palo Also Research Center, 1981.

[6] A. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. Technical Report CSL-83-7, Xerox Palo Also Research Center, 1983.

[7] Mary Hubley. Achieving Interoperability. Datapro Information Services Group. CW Custom Publications, Framingham, MA, 1994.

[8] John Dilley. Object-Oriented Distributed Computing with C++ and OSF DCE. International Workshop in DCE, October, 1993.

[9] Object Management Group. Common Object Request Broker Architecture and Specification. Document Number 91.12.1, Revision 1.1, 1991.

[10] Digital Equipment Corporation, Hewlett-Packard Company, Hyperdesk Corporation, International Business Machines Corporation, NEC Corporation, and Open Software foundation. Joint Submission on Interoperability and Initialization. OMG TC Document 94-3-5, March, 1994.